Network Working Group
Internet-Draft
Intended status: Experimental
Expires: September 14, 2017

D. Lopez
TID
J. Bonnet
Altice Labs
M. Peuster
UPB
P. Aranda Gutierrez
UC3M
March 13, 2017

The Role of a Mediation Element in NFV DevOps
draft-sonata-nfvrg-devops-gatekeeper-03

Abstract

This document describes how a mediation element (a "gatekeeper") can be applied to support DevOps practices in the provisioning of network services based on Network Function Virtualisation (NFV).

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

The DevOps model is already an established concept in IT industry reducing time to market by close collaboration between service developers and service operators. The switch to virtualisation technologies in the network and its potential for quicker time-to-market deployment requires the application of agile development cycles supporting a DevOps approach. This kind of approach will overcome key inhibitors that network operators face when deploying NFV, such as lack of legacy compatibility, resource orchestration, automation and multi-vendor interoperability, hence facilitating the transition to a software-driven network. The adoption of the DevOps model for network services will contribute to interaction between development, testing, and operation of network functionalities and network services. Both the function/service description formats as well as the infrastructure resource descriptions will be able to express and use legacy cases, e.g., the case of a non-virtual network function bound to a specific place in the network, with the data flows routed accordingly.

Network Service Providers (NSPs) must be able to orchestrate diverse network functions from multiple sources for automation and streamline them into an inter-organizational DevOps workflow. To embrace the DevOps model implies not only to shorten time between deploying, testing and validating of services, but also to enable the mechanisms for the network to consider application layer requirements and reaction to SLAs, and to ease network reconfiguration in order to achieve fast reaction in a timely manner.

Development and operational tools, the two essential pillars of DevOps, translate into the need of addressing the interfacing of service development tasks and the service platform, which in DevOps are closely linked together. It is required to emphasize the need for quick turn-around times in service development and operation, and materialize it in a mediated interface making a direct collaboration on both the development and the platform side possible.

The branching to multiple stakeholders in the service lifecycle creates an inter-organizational dynamics that must be taken into account. A realistic NFV DevOps approach has to take into account a trustworthy cycle with a mediation element that ensures compliance policies set by the NSP considering legacy situation, allowing developers across stakeholders to enter the ecosystem. Such a mediation element is what we will refer as a "gatekeeper" in the rest of this document. The resulting strategy opens collaborating opportunities while mitigating liability risks across the network service lifecycle.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying RFC-2119 significance.

3. The Essential Components for NFV DevOps

The collaboration between the development and operational tasks to build a service lifecycle according to the DevOps principles requires to combine service programming and orchestration frameworks by means of the following components:

o Catalogues, storing static information regarding network functions and services: code, executables, configuration data, and specific management requirements and preferences. Contents, location, organization, and implementation of catalogues for different artefacts can vary considerably. However, users of these catalogues need to deal with them in a consistent fashion and the differences across different catalogues need to be harmonized and abstracted away. As a high-level categorization, the following three types of catalogues can be considered:

   * Private catalogues of service developers, where they can define, access, reuse, and modify services and service components.

   * Service platform catalogues made available to authorized service developers for reusing existing components in their services, and used for storing services and their components that need to be deployed by the service platform.

   * Public catalogues storing artefacts developed and maintained by third-party developers on arbitrary platforms accessible to service developers and service platform operators.

o Service Development Kit (SDK). The SDK supports service developers by providing a service programming model and a development tool-chain, designed to support developers in defining and testing complex services consisting of multiple network functions, and to facilitate custom implementations of individual network functions. The tools of this tool-chains provides all necesaray interfaces to establish fully automated continuous

integration (CI) workflows. The implemented artefacts are stored in the developer's private catalogues. Moreover, service components can easily be obtained from external catalogues. The obtained artefacts can be directly used in a service or after being modified and tested using the SDK development tools. The service components and all the information necessary for deployment and execution of a service are bundled together into a package. The service package can be handed over to a service platform for actual deployment and for testing, debugging, and profiling purposes. The tools provided by a service development tool-chain can be classified as follows:

* Pre-validation tools used by service developers to ensure the correctness of service and function descriptions, including syntax checks, static structure validation, and integrity ensurance tools.

* Offline testing and emulation tools used by service developers to conduct functional tests of complex services in well-defined, reproducible environments. Especially focusing on the integration and interoperability between multiple network functions combined to a single service.

* Packaging tools responsible to create the final service bundle. This class includes tools that automatically resolve external dependencies of a service to automatically include required artifacts into a package. It also contains tools which sign the final package to give service platforms the necessary confidence that service packages have not been altered during the uploading procedure and to ensure the authenticity of the package.

* Tools to support the interaction of a developer with service platforms as well as services and functions deployed in these platforms. This includes two ways of interactions. First, uploading, instantiation and management of service packages on service platforms. Second, receiving runtime, debugging, and monitoring information from the platform as well as accessing artifacts stored in platform catalogues.

o Service Platform. The service platform receives the service packages implemented and created with the help of the SDK and is responsible for placing, deploying, provisioning, scaling, and managing the services on existing cloud infrastructures. It can also provide direct feedback about the deployed services to the SDK, for example, monitoring data about a service or its components. The service developer can ship the service package to the service platform together with service- or function-specific

lifecycle management requirements and preferences. A gatekeeper module in the service platform is responsible for processing the incoming and outgoing requests.

o Underlying Infrastructure. The infrastructure needs to host and execute the actual network functions of a service, e.g., as a virtual machine. The service platform sends necessary information and instructions for execution and lifecycle management of services to the infrastructure. The infrastructure may belong to the service platform operator, or to a third-party infrastructure operator. The interaction between the service platform and the infrastructure is done through a Virtual Infrastructure Manager (VIM). In a typical deployment, the service platform runs directly on top of an actual infrastructure. However, there can be service platforms supporting a recursive deployment model, where a service platform can act as an abstraction to the underlying infrastructure for another service platform. The service platform gatekeeper can play a relevant role to support mediated recursion as well.

The DevOps workflow is supported by the integration between the SDK and the service platform. This workflow implies continuous deployment and continuous integration during service development. The main entity exchanged between the SDK and the service platform is the service package to be deployed and runtime information like monitoring data and performance measurements regarding the service package, which is provided to the service developer during the development phase, as well as the runtime. This information can be used for optimizing, modifying, and debugging the operation and functionality of services.

4. The Role of the Gatekeeper

The gatekeeper is the module in the service platform that mediates the interactions between the SDK and the SP, settling the development and operational tasks, by performing the basic functions described here.

4.1. User Management

User management allows the service platform owner to control who can do what in the platform. This feature is particularly important in recursive scenarios, on which we may have a chain of service platforms interacting for the implementation of an end-to-end service.

The most basic feature of any user management component will be to

know who is the user, a feature that is usually called authentication. Authentication requires user registration and the maintenance of user identity attributes, including not only identification attributes (user identifiers, passwords, public keys, trusted signing certificates, etc.) but also other information supporting different authorization schemas, such as group-based or role-based ones.

The definition of what each (known) user can do is usually called authorization. The most common approach nowadays to authorization is called role-based, in which each user is assigned one (or more) role(s) and different roles have different permissions. This extra level of indirection, that is users to roles and roles to permissions, simplifies the overall maintenance of the system, when compared to a more direct scheme, like users permissions. Specially when accessing external APIs, it is common to issue temporary keys (then usually called tokens) which enable temporary access to those APIs. Real keys therefore do not leave the realm on which they are valid and useful, thus increasing the overall level of security.

To support a DevOps environment the following roles are considered:

o Developer, able to publish and update service packages on the service platform through the SDK, as well as other operations related to service package status.

o Service provider, in charge of structuring and managing the services available for a certain organization, or organizational group, defining an administrative domain.

o Customer, as a user of the public services available for the administrative domain they belong to, managing their lifecycles (instantiating, pausing, resuming, retiring...).

o Service platform admin, with management capabilities on the platform itself, as well as superuser-like control over the available services. These capabilities include the registration of roles and users, and the association of users to roles, enabling the authentication and authorization mechanisms described above.

4.2. **Package Management**

The gatekeeper receives the software to be validated in the form of packages. Package management is mostly about accepting and validating new or updated packages. The metadata describing such packages is called package descriptor, and constitutes the core of the gatekeeper interface.

Only known (i.e., successfully authenticated) and authorized users will be able to submit new or revised services through the gatekeeper. On-boarding of a package can only be considered successful when package validation and attestation is successful. Only then the (new version of) the package will become part of the catalogue. On-boarding requests are usually processed in a first come, first served way, otherwise contradictory requests may jeopardize the whole system. The usual solution for this problem is to use a queue mechanism that guarantees this sequence.

A package descriptor is validated in several ways:

- Syntax, comprising the validation against the expected package descriptor format.

- Semantics, which includes the validation of at least the basic parameters. The exact semantic aspects to be validated will depend on the content and format chosen for the package descriptor.

- Licensing, by checking that all external dependencies (i.e., packages, libraries or services) have to have their licenses checked before being used.

- Tests availability. Although this might be seen as part of the syntactic/semantic correction, there must be a set of tests that can be executed when validating the package. Depending of the scope and complexity of the service, these tests may be a subset of the unit tests or a more elaborate suit of integration tests.

- Tests execution. Besides providing a suit of tests, these have to be successfully executed. This execution may (usually will) imply the creation and initialization of at least one test environment. When the package under test depends on other packages, libraries or services, those too should be taken into account in the execution of the package tests.

The service package must include signatures, generated by the SDK's packaging tools, that allow the validation of the integrity and authenticity of the package's contents, the component VNFs, and other components (forwarding graphs, test suites, etc.). These signatured can be optionally used to attest the components at different stages of their lifecycle, and/or during runtime.

Requests for a change in the life-cycle of a package must be validated. This might be a simple authorization configuration.

- Deployment. Valid packages, available at the service platform repository, may receive a request for deployment. Package deployment implies the creation of all the environments and connections needed for the package and its dependencies to work and of an instance of that package.

- Instance (re)-configuration. A deployed package instance may need to be configured. A special kind of configuration might be, for packages supporting multi-tenancy, adding a new tenant. The package may have "open parameters" that can only be closed upon instantiation (e.g., an IP address). If a Package upgrade happens, a reconfiguration of the instance must also be made.

- Instance (re-)start. When, e.g., configuration changes.

- Instance monitoring. This is not strictly a change in the life-cycle, but would require the execution of certain aspects identified by the package descriptor or its components.

- Instance stop. Includes soft-stop (i.e., not accepting new requests and letting currently running request reach their end of life normally, with a pre-defined time-out) and hard-stop (i.e., a sudden stop, with requests still being answered by the service).

- Instance termination. Frees any resource(s) that were being used, taking care of dependencies.

- Removal. It requieres an evaluation of currently running instances and dependencies.

4.3. **Monitor Data Transfer**

The gatekeeper is the first point of access to reach the SP from the SDK. Service developers can use their identities from the SDK to access monitor data from the SP. After the successful AuthN/AuthZ phase, developers are granted a session token to access monitoring data. Multiple developers will use different data access views to get their own set of authorized monitor data.

It is desirable that the gatekeeper is transparent to the monitor data transfer, acting as a pure forwarder, apart from the AuthN/AuthZ phase. Optionally, the gatekeeper could filter non-numerical monitored data (e.g. obfuscate domain names, IP/MAC addresses...) transferred in logfiles or packet streams. The session token is used by the monitor data management components to decide on which data to expose, so metrics of another user, other services not started by the developer or the SP itself can never be queried by the SDK. In addition, other limits can be enforced, such as:

- Limit the number of monitor samples

- Limit the data size to be received

- Limit the time frame during which metrics are accessible

5. Security Considerations

The gatekeeper acts as the security enforcement point for all DevOps interactions between the development and operational tasks, and even between different layers in recursive structure.

Gatekeeper APIs will have to be secured, providing identification, confidentiality, integrity and non-repudiation.

Other potential threats are related to denial-of-service, whereby an adversary could make the whole NFV environment unusable by overloading the gatekeeper with a high number of requests or requests tailored to exhaust its resources. Mechanisms for overload detection and mitigation should be put in place.

6. IANA Considerations

This document requires no IANA actions.

7. Acknowledgements

This work has been partially performed in the scope of the SONATA project [SONATA], which has received funding from the European Union's Horizon 2020 research and innovation programme. The authors would like to acknowledge the contributions of their colleagues. This information reflects the consortium's view, but the consortium is not liable for any use that may be made of any of the information contained therein.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

8.2. Informative References

[SONATA]   "Project SONATA", <http://www.sonata-nfv.eu/>.

Authors' Addresses

Diego R. Lopez
Telefonica I+D
Zurbaran, 12
Madrid,   28010
Spain

Phone: +34 913 129 041
Email: diego.r.lopez@telefonica.com

Jose Bonnet
Altice Labs
Rua Eng. Jose Ferreira Pinto Basto
Aveiro,   3810-106
Portugal

Phone: +351 234 403 200
Email: jbonnet@alticelabs.com

Manuel Peuster
Paderborn University
Warburgerstrasse 100
Paderborn,   33098
Germany

Phone: +49 5251 60 4341
Email: manuel.peuster@upb.de

Pedro A. Aranda Gutierrez
UC3M

Email: paaguti@hotmail.com